Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

MAGISTER THESIS

Jakub Adámek

Neural Networks Controlling Prosody of Czech language

Department of software engineering

Supervisor: *Mgr. Roman Neruda, CSc.*

Computer Science

I would like to thank to Roman Neruda for guiding the works on the thesis, Jirka Hanika for giving me all the advise needed and Petr Horák for preparing the speech corpus. Last but not least thanks to Eva for all support.

I have written this Thesis by myself using only the sources cited. Lending the Thesis is allowed.

Prague, 19.4.2002 Jakub Adámek

Contents

Abstrakt

Název práce: Neuronové sítě řídící prozódii české řeči

Autor: Jakub Adámek

Katedra (ústav): Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Roman Neruda, CSc.

E-mail vedoucího: roman@cs.cas.cz

Abstrakt: Práce popisuje teoretické základy a programové prostředí pro učení neuronových sítí perceptronového typu, včetně rekurentní sítě pro časové řady. Implementovány jsou dva pokročilé učící algoritmy, jeden ze skupiny algoritmů sdruženého gradientu, druhý Levenberg-Marquardtův s určitým vylepšením. Síť je pak použita k řešení praktického problému generování prozódie v systému Epos pro syntézu řeči z psaného textu. Inspiraci poskytuje systém pro syntézu řeči používající neuronovou síť v němčině. V programu je kladen důraz na snadné definování nových vstupů sítě. Pomocí malého korpusu je naučena konkrétní neuronová síť generující prozódii v českých oznamovacích větách čtených neutrálním hlasem.

Klíčová slova: neuronové sítě, perceptron, syntéza řeči, prozódie

Abstract

Title: Neural Networks Controlling Prosody of Czech language

Author: Jakub Adámek

Department: Department of Software Engineering

Supervisor: Mgr. Roman Neruda, CSc.

Supervisor's e-mail address: roman@cs.cas.cz

Abstract: This work describes a theoretical basis and a program environment for learning perceptron-based neural networks, including recurrent ones for time series. Two advanced learning techniques are implemented, one from the group of conjugate gradients, the other being Levenberg-Marquardt's with some improvement. The network is than used to solve a practical problem of prosody generation in a text-to-speech system Epos. An inspiration is gained from other text-to-speech system using neural networks in German. The program emphasizes an easy definition of new network inputs. A particular network generating prosody of Czech indicative sentences read in a neutral voice is trained with a small corpus and its results are evaluated.

Keywords: neural networks, perceptron, text-to-speech, prosody

Goal and scope of this thesis

The present thesis has been initiated by the need of prosody generation formed by a neural network in the Text-To-Speech (TTS) system Epos. There was some very simple neural network support in the system already, which was never used in practice. The current prosody in Epos is formed by rules. There was a hope of better results and an interest in any results from a neural network. The problem of prosody generation involves time series prediction — prosody generated for syllables must take into account the neighbour syllables. Thus the recurrent network structure had to be supported.

The goal was to prepare an environment allowing to intelligibly describe inputs to a neural network and the usage of its outputs so that people who do not understand programming but do understand linguistic may use it. And to create another environment in which the neural network learns. Also a particular network for Czech indicative sentences read in a neutral voice was to be prepared.

A parallel goal was to provide neural network support in the multi-agent system for artificial intelligence Bang 3, created at the Institute of Computer Science, Czech Academy of Science, Prague. The neural networks module should implement the perceptron networks and their training algorithms. The Bang system contains interchangeable agents, which means the perceptron network agent may be replaced by other network architectures and other agents may provide other learning algorithms, genetic ones for instance. In the Bang 2 system the Radial Basis Function networks were implemented by Petra Kudová [10] and will be re-implemented in Bang 3.

It is necessary to clearly state what are my own contributions in this thesis. All the C++ source files concerning neural networks are written by myself. A smaller part of the basic data structures and some XML read / write procedures are written by myself. The Bang environment with everything concerning agents and their communication was written by others. The whole Epos system was written by others, I have added a new prosody rule "neuralnet" and the code working with its configuration file. The code written by myself is described in more detail in Appendix B.

This thesis is divided into two parts. The Neural Networks part describes thoroughly the perceptron networks with some advanced learning algorithms and other details concerning generalization improvement, training data and training process. The Radial Basis Function networks are roughly described and compared with the perceptron ones. This part contains the main points of the neural network successfully used in the Swiss TTS system SVOX. The Practical part describes the basics of the Epos and Bang 3 systems. It tells about the agents created and about the neural network usage in Epos and describes the particular network included with this thesis.

Part I

Neural Networks

Chapter 1

Introduction

The Neural Networks part starts with a thorough description of the perceptron networks. The Radial Basis Function networks are than roughly pictured and compared with the perceptron ones. Next chapter is concerned with the learning algorithms, explains the details related to gradient computation, weights initialization and error functions and presents two advanced learning algorithms. The problem of generalization and recurrent networks for time series prediction are mentioned. Next chapters tell how to work with training data and how does the training process run. The last chapter describes two neual networks used in a Korean and a German text-to-speech system.

1.1 Perceptron network

The perceptron and RBF network models are used to approximate a function $f : I \to O, I \subseteq \mathbb{R}^n, O \subseteq \mathbb{R}^m$. The network is trained by examples of $x \in I$ and the corresponding $f(x)$. When the example data cover the input region I enough, the resulting network is able to more or less accurately approximate the values for inputs which it has never seen. The perceptron neural network model is used in 80% of all the neural networks' applications [1, p. 52].

Let us explain the idea of perceptron networks on an example of a feed-forward two-layer neural network. The perceptron units are organized into layers. Feed-forward means the network is acyclic and usually also that connections lead only from the neurons on the layer i to the neurons on the layer $i+1$ and all the neurons on the layer i are connected to all the neurons on the layer $i+1$. The inputs are considered as the layer 0. The number of layers includes the output layer and does not include the input layer. The layers between the input and the output layer are called hidden layers, a

two-layer network has one hidden layer.

The network contains units called perceptrons. Each perceptron has its inputs $p_1, \ldots, p_R$ and a threshold which sets its sensibility — some neuron transfer functions discriminate well only in a region around 0 and are nearly constant for numbers with a big absolute value. The negative of the threshold $b = -$threshold is called bias. A perceptron activation n is a sum

$$n = b + \sum_{i=1}^{R} w_i p_i$$

and a perceptron's output is

$$o = f(n) = f\left(b + \sum_{i=1}^{R} w_i p_i\right) \tag{1.1}$$

A transfer function f is applied to the activation to get the output. The most usual ones are:

- Linear function
$$\text{purelin}(n) = n$$
- Hard limit, which is also called step activation.
$$\text{hardlim}(n) = \begin{cases} 0 & \text{when } n < 0, \\ 1 & \text{otherwise} \end{cases}$$
- Logical sigmoid, which continuously approximates the hard limit. The parameter λ is often set to 1. Jiroušek [8] experimented with the λ adaptation and concluded not to use other values than 1.
$$\text{logsig}(n) = \frac{1}{1 + e^{-\lambda n}}$$
- Hyperbolic tangent sigmoid — the second expression is computationally much faster as it contains only one e^x computation:
$$\tanh(n) = \frac{e^n - e^{-n}}{e^n + e^{-n}} = \frac{2}{1 + e^{-2n}} - 1$$

When using the backpropagation method, the transfer function must be derivable. All except the hard limit are. Usually the logical sigmoid is used in the hidden layers and the linear function or the logical sigmoid in the output layer depending on the desired output interval.

The network works in steps. First it calculates outputs for the neurons in the first layer. The output vector $\boldsymbol{a}^1$ is the input for the next layer:

$$a_i^1 = f_i\left(b_i + \sum_{j=0}^{n_0-1} w_{ji}p_j\right), \quad i = n_0, \ldots, n_1 - 1, \tag{1.2}$$

where n_0 is the number of inputs, $n_i - n_{i-1}$ number of neurons in the i-th layer, w_{ji} the weight of the connection from neuron j to neuron i, p_j the j-th input.

The network calculates the outputs of the second layer in the same way and these are the outputs $\boldsymbol{o}$ of the whole network:

$$o_i = f_i\left(b_i + \sum_{j=n_0}^{n_1-1} w_{ji}a_j^1\right), \quad i = n_1, \ldots, n_2 - 1 \tag{1.3}$$

More general architectures like more hidden layers or connections between the neurons on the same layer are sometimes considered. The basic requirement is the network being feed-forward, i.e. acyclic — there must be an order that allows to know the outputs of all the neurons connected to a neuron at the moment we compute its output.

1.2 Radial Basis Function network

Another class of the neural network model are the Radial Basis Function (RBF) networks [2]. The activation of a hidden unit is determined by the distance (usually Euclidean) between the input vector $\boldsymbol{x}$ and a prototype vector $\boldsymbol{\mu}_j$, the output is $\phi(\|\boldsymbol{x} - \boldsymbol{\mu}_j\|)$, where ϕ is some non-linear function, the most common being the Gaussian

$$\phi(x) = \exp\left(-\frac{x^2}{2\sigma^2}\right), \tag{1.4}$$

where σ is a parameter whose value controls the smoothness properties of the function, $\exp(x)$ means e^x.

The output of the whole network is a linear combination of the basis functions plus a bias w_{k0}

$$y_k(\boldsymbol{x}) = \sum_{j=1}^{M} w_{kj}\phi_j(\boldsymbol{x}) + w_{k0}, \tag{1.5}$$

where M is the number of radial basis units and w_{kj} the weight of the connection between the j-th unit and the k-th output.

For the case of the Gaussian basis function we have

$$\phi_j(\boldsymbol{x}) = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{\mu}_j\|}{2\sigma_j^2}\right), \tag{1.6}$$

where $\boldsymbol{x}$ is an input vector and $\boldsymbol{\mu}_j$ is the vector determining the center of the basis function ϕ_j.

More general topologies, like more than one hidden layer, are not normally used. A key aspect of the radial basis function networks is the distinction between the roles of the first and second layers of weights [2, p. 170]. This leads to a two-stage training procedure. In the first stage the parameters governing the basis functions (e.g. $\boldsymbol{\mu}_j$ and σ_j) are determined by unsupervised training techniques from the input data set $\{\boldsymbol{x}^n\}$ alone. In the second stage the basis functions are kept fixed while the second-layer weights are found. We can split the first stage into finding the unit centers $\boldsymbol{\mu}_j$ and finding the remaining parameters (like σ_j) [10].

More details about the RBF networks and their training may be found in Kudová [10].

1.3 Comparison

Both RBF and perceptron networks approximate arbitrary non-linear functional mappings between multidimensional spaces. But the structures of the networks are very different. Some of the important differences are [2, p. 82]:

- The perceptrons divide the space into a number of subspaces by hyperplanes. The radial basis units use distance to a prototype vector with a usually localized function.

- A perceptron network forms a distributed representation in the space — many hidden units typically contribute to the output value. The interference between the hidden units results in a highly non-linear training process with problems of local minima. By contrast, only a few radial basis units with localized basis functions have significant activations and determine the output.

- Perceptron networks have more layers of weights and a complex pattern of connectivity, not all the possible weights are usually present. Also, the activation functions may differ for each neuron. An RBF network, however, generally has a simple architecture described above.

- All the parameters of perceptron networks are determined at once, while the RBF network is typically trained in a much faster two stage technique. Although gradient learning similar to the steepest gradient descent (see p. 11) is used successfully for RBF networks, too [10].

Chapter 2

Perceptron network learning algorithms

The learning goal is the network approximates the outputs for any inputs from given range. We provide examples to the training algorithm — inputs and desired outputs. All the range must be covered with examples, otherwise there is no chance to achieve good performance.

Although it looks like that, our primary goal is not to learn the network to answer correctly to the training inputs. We want that it is capable of generalization — that it answers correctly to inputs which were not a part of training.

In the basic algorithm we try to improve the error function in the steepest direction — negative of the gradient. We hope that after finding the minimum, the error gradient will vanish and we will stop. Unfortunately there is a risk to get stuck in a local minima or to step over the real minima and search elsewhere. Various improvements of the basic backpropagation algorithm deal with these problems.

This and other gradient methods form a very broad and perhaps the most important class of numerical optimization methods. Four basic groups of them are: the steepest descent, the conjugate gradient, quasi-Newton and Newton methods.

Newton optimization method is computationally the most complex, however the most efficient with much higher convergence rate than other gradient methods. It uses the second partial derivatives. But, as Newton algorithm does not guarantee convergence if the initial values of the network parameters are relatively poor, it is usually used in a slightly modified form. Depending on performed modifications various, so called Newton-type, algorithms are obtained. The most often used quasi-Newton-type algorithm is the Levenberg-Marquardt algorithm.

Further I will describe the basic gradient descent and two advanced algorithms: one of the conjugate gradient family and the other a Levenberg-Marquardt algorithm modification.

2.1 Weights initialization

Initial weights should be our first guess of how the network should look. But usually we use some simple method. We can set them to random small numbers, e.g. from $[-1; +1]$ or in a more sophisticated way about $1/\sqrt{|j_{\leftarrow}|}$, where $|j_{\leftarrow}|$ is the number of neurons connected to j [2, p. 262].

Nguyen and Widrow proposed a method for selecting the initial weights so that they are distributed over all the space [14]. The weights are set for each neuron this way: Suppose all neuron inputs have average 0 and standard deviation 1 and the transfer function has the active region $[-1; +1]$. That means for values outside the active region the result will be almost 1 or almost -1. For example for the logical sigmoid function inputs from $[-3; +3]$ are in the interval $[0.05; 0.95]$ hence $[-3; +3]$ may be considered the active region. Set

$$W' = \kappa \sqrt[p]{m} \tag{2.1}$$

where κ is the overlapping factor usually set to 0.7, m is count of neurons on the same level and p is count of neurons on previous level or inputs.

Prepare p random numbers $a_i \in [-1; +1]$. The weights on connections coming to our neuron are normalized by the norm of the vector $\boldsymbol{a}$.

$$w_i = W' \frac{a_i}{|\boldsymbol{a}|} \tag{2.2}$$

The bias weight is chosen randomly from $[-1 - w_1; 1 + w_1]$. If the neuron inputs or the transfer function active region are distributed over another interval, transform the formulas appropriately.

2.2 Error function

An error measure $E(\boldsymbol{w})$ needs to be defined as a function of the weight vector $\boldsymbol{w}$ of the network. The most common one is the sum-squared error (SSE)

$$E^q(\boldsymbol{w}) = \frac{1}{2} \sum_{j \in Y} (o_j^q - d_j^q)^2 \tag{2.3}$$

$$E_{\mathrm{SSE}}(\boldsymbol{w}) = \sum_{q=0}^{p-1} E^q(\boldsymbol{w}) = \frac{1}{2} \sum_{q=0}^{p-1} \sum_{j \in Y} (o_j^q - d_j^q)^2, \tag{2.4}$$

where o_j means output calculated by the network, d_j the desired output, Y the set of output neurons. It is useful to normalize the error function by the count of training rows — a root mean square error is

$$E_{\mathrm{RMS}}(\boldsymbol{w}) = \frac{1}{p} \sum_{q=0}^{p-1} E^q(\boldsymbol{w}). \tag{2.5}$$

We can try to force the "useless" weights to become zero. This can be done by adding a penalty term to the error function, e.g.

$$E = E_{\mathrm{SSE}} + \frac{b}{2} \sum_{i \in j_{\leftarrow}} w_{ij}^2 \tag{2.6}$$

This is equivalent to having a weight decay term in the steepest gradient descent algorithm (see p. 11)

$$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}} + b w_{ij} \tag{2.7}$$

The decay rate b is initialized with zero and then gradually increased to some value. This decay term will pull the weights to zero values. Only those constantly modified by errors coming form the patterns i.e. only the essential weights will "survive".

2.3 Computing gradient — backpropagation

The function computed by the network is very difficult and it is too expensive to compute the error gradient directly. But it is easy to get it for the output neurons. And having the value for all the neurons to which a particular neuron is connected, we can compute the error function partial derivative for that neuron. This way we backpropagate it from the outputs to the inputs.

Let us index the neurons over all the layers to simplify the description. The inputs have indexes $0, \ldots, n_0 - 1$, neurons in i-th layer $n_{i-1}, \ldots, n_i - 1$, where n_0 is the input count, $n_i - n_{i-1}$ is the count of neurons in i-th layer and L the layer count. Let us consider bias as a neuron with index -1, with a constant output 1. Let us denote $j_{\leftarrow}$ the set of all the neurons, which are connected to j and $j^{\rightarrow}$ all the neurons, to which is j connected.

The weights form a vector $\boldsymbol{w} = (w_{ij} | i \in -1, \ldots, n_{L-1} - 1, j \in i^{\rightarrow})$, where w_{ij} is the weight of the connection between neurons i and j.

The training data set is $T = \{(\boldsymbol{x}^q, \boldsymbol{d}^q) \mid q = 0, \ldots, p-1\}$, where $\boldsymbol{x}^q$ are inputs and $\boldsymbol{d}^q$ desired outputs. The set of input neurons is X and the set of output neurons is Y. Outputs computed by the network for the training example q are $\boldsymbol{o}^q$, where $o_j^q = f_j(n_j^q)$, f_j is the transfer function for $j \in Y$ and $n_j^q = \sum_{r \in j_\leftarrow} a_r^q w_{rj}$.

At the start of the training process we need to guess initial weights — see p. 8. Using the chain rule, the gradient may be expressed as

$$\frac{\partial E^q}{\partial w_{ij}} = \frac{\partial E^q}{\partial a_j} \frac{\partial a_j}{\partial n_j} \frac{\partial n_j}{\partial w_{ij}}, \tag{2.8}$$

$$\text{where } \frac{\partial n_j}{\partial w_{ij}} = a_i.$$

The partial derivative of the transfer function $\frac{\partial a_j}{\partial n_j}$ is 1 for the linear function. For the logical sigmoid it is

$$\frac{\partial a_j}{\partial n_j} = \frac{\lambda_j e^{-\lambda_j n_j}}{(1 + e^{-\lambda_j n_j})^2} = \lambda_j a_j (1 - a_j) \tag{2.9}$$

and for the hyperbolic tangent sigmoid it is

$$\frac{\partial a_j}{\partial n_j} = 1 - a_j^2 \tag{2.10}$$

We need to count the last partial derivative $\delta_j^q = \frac{\partial E^q}{\partial a_j}$. Here we use the backpropagation strategy. For an output neuron $j \in Y$ we may directly derive the error function:

$$\delta_j^q = \frac{\partial E^q}{\partial a_j} = \frac{\partial a_j}{\partial n_j}(o_j^q - d_j^q) \text{ for } j \in Y. \tag{2.11}$$

For a hidden neuron we can use the chain rule again:

$$\delta_j^q = \frac{\partial E^q}{\partial a_j} = \sum_{r \in j^\rightarrow} \frac{\partial E^q}{\partial a_r} \frac{\partial a_r}{\partial n_r} \frac{\partial n_r}{\partial a_j} = \frac{\partial a_j^q}{\partial n_j} \sum_{r \in j^\rightarrow} \delta_r^q w_{rj} \text{ for } j \notin Y \tag{2.12}$$

This way we have transformed the computation to the neurons to which the neuron j is connected and the values of which we already know, because we move from the output layer to the input one. This method is correct when the network is acyclic. The backpropagation is summarized as the algorithm 1.

There are two different strategies of the weights update. In the batch approach first the error for all the training examples is counted and only

Algorithm 1: Backpropagation

initialize the weights (see p. 8)
for all training inputs $\boldsymbol{x}^q$
    apply $\boldsymbol{x}^q$ to the input layer
    propagate $\boldsymbol{x}^q$ forward to the output layer, using eq. 1.1
    calculate error $E^q(\boldsymbol{w})$ according to eq. 2.3
    compute the δ's of the output layer as in eq. 2.11
    compute the δ's of the preceding layers, by backpropagating the δ's backward as in eq. 2.12

than (at the end of an epoch) the weights are changed. In the incremental approach the examples are presented usually in a random order and the weights are changed after each training example.

The simplest method called steepest gradient descent changes the weights at every epoch proportionally to the negative of the gradient of $E(\boldsymbol{w})$,

$$\Delta w_{ij}^q = -\eta \frac{\partial E^q}{\partial w_{ij}} \tag{2.13}$$

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = \sum_{q=0}^{p-1} \Delta w_{ij}^q, \tag{2.14}$$

where η is a small positive number $0 < \eta < 1$ called learning rate. The learning process is extremely sensible to the learning rate. If a too big one is used, it misses the error function minimum and never converges. If a too small one is used, the training lasts too long.

There are many improvements to this basic method, like the momentum term or an automatical learning rate adjustment. I did not implement any of them because other training methods described in this thesis are much better even with all the improvements.

2.4 Weight perturbation

An alternative numerical approach for computing the derivatives of the error function is to use finite differences [2, p. 147]. This can be done by perturbing each weight in turn, and approximating the derivatives

$$\frac{\partial E^n}{\partial w_{ij}} = \frac{E^n(w_{ij} + \epsilon) - E^n(w_{ij})}{\epsilon} + O(\epsilon), \tag{2.15}$$

where $\epsilon \ll 1$ is a small number, e.g. 0.00001. This method is computationally very expensive — we have to run the network on all training data for each weight. The main usage is to verify the correctness of the backpropagation algorithm, which is essential to the learning techniques. The weights perturbation technique is extremely easy both to understand and to implement.

A big improvement in the terms of computing time to the weight perturbation may be done by the node perturbation [2]. Remember the equation 2.8

$$\frac{\partial E^n}{\partial w_{ij}} = \frac{\partial E^n}{\partial a_j} \frac{\partial a_j}{\partial n_j} \frac{\partial n_j}{\partial w_{ij}}.$$

We can perturb only the neuron activations a_j

$$\frac{\partial E^n}{\partial a_j} = \frac{E^n(a_j + \epsilon) - E^n(a_j - \epsilon)}{2\epsilon} + O(\epsilon) \tag{2.16}$$

and count the remaining two derivatives $\frac{\partial a_j}{\partial n_j}$ and $\frac{\partial n_j}{\partial w_{ij}}$ directly, which is easy. Here we need to run the network on all the training patterns for each neuron, which is much less than the number of weights. However it is always much slower than the backpropagation, which is run only twice on the training patterns — one forward and one backward run. The node perturbation is a bit more difficult than the weight perturbation and therefore less suitable for verifying the backpropagation.

2.5 Stable dynamic parameter adaptation

Rüger [18] proposes a request for minimization: asymptotic stability, i.e. ensuring that the performance of the parameters does not decrease at the end of learning. This stability criterion allows for greedy steps in the initial phase of learning, although the error does not decrease at every step.

He proposes a class of stable algorithms and proves the asymptotic convergence for them. What is more, he shows these algorithms provide a significant improvement in the speed of the training process.

The inspiration of this algorithms comes from Salomon [16, 17], who proves the setting of a new parameter ζ (see below) is uncritical: all values work, especially sensible ones being those between 1.2 and 2.1.

Rüger's definition says [18, p. 3]:

"Let $E : \mathbb{R}^n \to \mathbb{R}$ be an error function of a neural net with random weight vector $\boldsymbol{w}^0 \in \mathbb{R}^n$. Let $\zeta > 1, \eta_0 > 0, 0 < c \leq 1,$ and $0 < a \leq 1 \leq b$. At step t of the algorithm, choose a vector $\boldsymbol{g}^t$ restricted only by the conditions

$$\frac{\boldsymbol{g}^t \nabla E(\boldsymbol{w}^t)}{\|\boldsymbol{g}^t\| \|\nabla E(\boldsymbol{w}^t)\|} \geq c \tag{2.17}$$

and that it either holds for all t that $1/\|\boldsymbol{g}^t\| \in [a;b]$ or that it holds for all t that $\|\nabla E(\boldsymbol{w}^t)\|/\|\boldsymbol{g}^t\| \in [a;b]$, i.e., the vectors $\boldsymbol{g}^t$ have a minimal positive projection onto the gradient and either have a uniformly bounded length or are uniformly bounded by the length of the gradient. Note that this is always possible by choosing $\boldsymbol{g}^t$ as the gradient or the normalized gradient.

Let $e : \eta \mapsto E(\boldsymbol{w}^t - \eta \boldsymbol{g}^t)$ denote a one-dimensional error function given by E, w^t and g^t. Repeat (until the gradient vanishes or an upper limit of t or a lower limit E_min of E is reached) the iteration $\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta_{t+1}\boldsymbol{g}^t$ with:

$$\eta_{t+1} = \begin{cases} \eta^* := \dfrac{\eta_t\zeta/2}{1 + \dfrac{e(\eta_t\zeta) - e(0)}{\eta_t\zeta \boldsymbol{g}^t \nabla E(\boldsymbol{w}^t)}} & \text{if } e(0) < e(\eta_t\zeta) \\ \eta_t/\zeta & \text{if } e(\eta_t/\zeta) \leq e(\eta_t\zeta) \leq e(0) \\ \eta_t\zeta & \text{otherwise} \end{cases} \tag{2.18}$$

The first case for η_{t+1} is a stabilizing term η^*, which definitely decreases the error when the error surface is quadratic, i.e., near a minimum. η^* is put into effect when the error $e(\eta_t\zeta)$, which would occur in the next step if $\eta_{t+1} = \eta_t\zeta$ was chosen, exceeds the error $e(0)$ produced by the present weight vector $\boldsymbol{w}^t$. By construction, η^* results in a value less than $\eta_t\zeta/2$ if $e(0) < e(\eta_t\zeta)$; hence, given $\zeta < 2$, the learning rate is decreased as expected, no matter what E looks like. Typically (if the values for ζ are not extremely high) the other two cases apply, where $\eta_t\zeta$ and η_t/ζ compete for a lower error.

Note that, instead of gradient descent, this class of algorithms proposes a $\boldsymbol{g}^t$ descent, and the vectors $\boldsymbol{g}^t$ may differ from the gradient. A particular algorithm is given by a specification of how to choose $\boldsymbol{g}^t$."

Rüger suggests to incorporate the Polak-Ribiere rule,

$$\boldsymbol{d}^{t+1} = \nabla E(\boldsymbol{w}^{t+1}) + \alpha\beta\boldsymbol{d}^t, \tag{2.19}$$

for conjugate directions with $\boldsymbol{d}^0 = \nabla E(\boldsymbol{w}^0), \alpha = 1$ (by choosing $\alpha = 0$, one gets an algorithm similar to the Salomon's one), and

$$\beta = \frac{(\nabla E(\boldsymbol{w}^{t+1}) - \nabla E(\boldsymbol{w}^t))^T \nabla E(\boldsymbol{w}^{t+1})}{\|\nabla E(\boldsymbol{w}^t)\|^2} \tag{2.20}$$

to propose vectors $\boldsymbol{g}^t := \boldsymbol{d}^t/\|\boldsymbol{d}^t\|$. One should reset the direction $\boldsymbol{d}^t$ after each n (the number of weights) updates to the gradient direction. Another reason for resetting the direction arises when $\boldsymbol{g}^t$ does not have the minimal positive projection c onto the normalized gradient. The method is summarized as the algorithm 2.

Algorithm 2: Stable conjugate-gradient descent

set $\zeta \in [1.2; 2.1], \eta_0 > 0, c \in (0; 1]$
set $\alpha = 1$ (or $\alpha = 0$ to obtain an algorithm very similar to Salomon's)
$t = 0$
while goal not achieved (see p. 24)
    calculate $\nabla E(\boldsymbol{w}^t)$ by standard error backpropagation
    if $t \equiv 0 \pmod{n}$ (n means number of weights) **then**
        $\boldsymbol{d}^t = \nabla E(\boldsymbol{w}^t)$
    else
        calculate β as in eq. 2.20
        $\boldsymbol{d}^t = \nabla E(\boldsymbol{w}^t) + \alpha\beta\boldsymbol{d}^{t-1}$
        if $\boldsymbol{g}^t = \boldsymbol{d}^t/\|\boldsymbol{d}^t\|$ does not satisfy the eq. 2.17 **then**
            $\boldsymbol{d}^t = \nabla E(\boldsymbol{w}^t)$
    store the weights $\boldsymbol{w}^t$
    calculate $e(x) = E(\boldsymbol{w}^t - x\boldsymbol{g}^t)$ at the points $x = 0, x = \eta_{t-1}\zeta, x = \eta_{t-1}/\zeta$
    choose η_t by eq. 2.18
    change the weights $\boldsymbol{w}^t = \boldsymbol{w}^{t-1} - \eta_t\boldsymbol{d}^t/\|\boldsymbol{d}^t\|$
    $t = t + 1$

2.6 Modified Levenberg-Marquardt

Steepest descent learning algorithms are based on linear approximation of the performance function, Newton algorithm on its quadratic approximation [14].

Let us rewrite the performance (error) function as

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{q=0}^{p-1}(\boldsymbol{e}^q)^T\boldsymbol{e}^q = \frac{1}{2}\boldsymbol{e}^T\boldsymbol{e}, \tag{2.21}$$

where $\boldsymbol{e}^q = \boldsymbol{e}^q(\boldsymbol{w})$ is the error vector for the q-th data pair from the data set $T = \{(\boldsymbol{x}^q, \boldsymbol{d}^q) \mid q = 0, \ldots, p-1\}$ (we sometimes do not write the parameter $\boldsymbol{w}$), $e_i^q(\boldsymbol{w}) = d_i^q - o_i^q(\boldsymbol{w})$, $\boldsymbol{o}^q$ is the network output, $\boldsymbol{e}$ is the error vector for the whole data set with dimensions $n(e) = |Y||T|$, Y is the set of output neurons.

Newton algorithm for minimization of the performance function $E(\boldsymbol{w})$ is

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^k - \boldsymbol{H}^{-1}\nabla E(\boldsymbol{w}^k), \tag{2.22}$$

where k is the epoch number, $\boldsymbol{H}(\boldsymbol{w}) = \nabla^2 E(\boldsymbol{w})$ is the Hessian matrix of second derivatives and $\nabla E(\boldsymbol{w})$ is the gradient of the performance (error) function. If the performance function is defined by eq. 2.21 it can be written:

$$\nabla E(\boldsymbol{w}) = \boldsymbol{J}^T(\boldsymbol{w})\, \boldsymbol{e}(\boldsymbol{w}), \tag{2.23}$$

$$\boldsymbol{H} = \nabla^2 E = \boldsymbol{J}^T \boldsymbol{J} + \sum_{i=1}^{n(e)} e_i(\boldsymbol{w}) \nabla^2 e_i(\boldsymbol{w}), \tag{2.24}$$

where $\boldsymbol{J}(\boldsymbol{w})$ is the Jacobian matrix of first derivatives, which can be easy computed by backpropagation.

$$\boldsymbol{J}(\boldsymbol{w}) = \frac{\partial \boldsymbol{e}(\boldsymbol{w})}{\partial \boldsymbol{w}} = \begin{bmatrix} \dfrac{\partial e_1(\boldsymbol{w})}{\partial w_1} & \cdots & \dfrac{\partial e_1(\boldsymbol{w})}{\partial w_{n(w)}} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial e_{n(e)}(\boldsymbol{w})}{\partial w_1} & \cdots & \dfrac{\partial e_{n(e)}(\boldsymbol{w})}{\partial w_{n(w)}} \end{bmatrix}, \tag{2.25}$$

The summation element in equation 2.24 may be costly to compute and it will be close to zero when the output error becomes small. If it is neglected Gauss-Newton algorithm is obtained

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^k - \left[\boldsymbol{J}^T(\boldsymbol{w}^k)\boldsymbol{J}(\boldsymbol{w}^k)\right]^{-1} \nabla E(\boldsymbol{w}^k) \tag{2.26}$$

Even though this expression is ensured to be positive semidefinite [14], it may be singular or close to singular. This is the case, for example, if the neural network is overparametrized or the data are not informative enough. Various ways to overcome this problem exist and are known as regularization techniques. One of them is the Levenberg modification

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^k - \left[\boldsymbol{J}^T(\boldsymbol{w}^k)\boldsymbol{J}(\boldsymbol{w}^k) + \mu^k \boldsymbol{I}\right]^{-1} \nabla E(\boldsymbol{w}^k) \tag{2.27}$$

where μ is the regularization parameter. Bishop says [2, p. 292]: "For very small values of the parameter μ we recover the Newton formula, while for large values of μ we recover standard gradient descent."

Marquardt developed a very efficient, the most often used procedure for adjusting the parameter μ. In the k-th epoch an optimal value μ^k is determined by an iterative procedure (Algorithm 3).

The adjustment of the regularization parameter μ starts by multiplying with a decreasing factor $\mu_d < 1$. If it results in an increase in the performance function, μ is multiplied by an increasing factor $\mu_i > 1$ as many times as necessary. Marquardt suggested $\mu_d = 0.1, \mu_i = 10$ and $\mu^0 = 0.001$.

Algorithm 3: Levenberg-Marquardt

$\mu^k = \mu^{k-1}\mu_d$
calculate new values $\boldsymbol{w}^{k+1}$ by eq. 2.27, $E(\boldsymbol{w}^{k+1})$ by eq. 2.4
while $E(\boldsymbol{w}^{k+1}) \geq E(\boldsymbol{w}^k)$ **do**
    $\mu^k = \mu^k\mu_i$
    calculate new values $\boldsymbol{w}^{k+1}$, $E(\boldsymbol{w}^{k+1})$

Algorithm 4: Modified Levenberg-Marquardt

if $E(\boldsymbol{w}^k) < (1-h)E(\boldsymbol{w}^{k-1})$ **then**
    $\mu^k = \mu_d\mu^{k-1}$
    $E_{\min} = E(\boldsymbol{w}^k)$
else if $E(\boldsymbol{w}^k) \leq (1+h)E(\boldsymbol{w}^{k-1})$ **or** $E(\boldsymbol{w}^k) \geq E_{\min}$ **then**
    $\mu^k = \mu^{k-1}$
else
    $\mu^k = \mu_i\mu^{k-1}$

Another variant of the Levenberg-Marquardt algorithm is proposed in [14]. It differs in the calculation procedure of μ, which is a simple, one step procedure (algorithm 4).

The algorithm convergence is ensured if $\mu_d\mu_i > 1$. The parameter h cares about decreasing the oscillations of the performance function near its minimum. The authors used values $\mu_d = 0.6, \mu_i = 2, \mu^0 = 0.001, h = 0.005$. According to them, the modification shows better performance regarding probability of escape from a local minima thanks to its oscillations. Examples of different learning tasks are presented, which show the modified algorithm is faster.

2.7 Methods comparison

None of the methods is the best one, for different tasks different methods have best results [3, p. 5-49]. The Levenberg-Marquardt (LM) algorithm is excellent on function approximation problems with smaller networks with a few hundreds of weights, especially when we need a big accuracy. However, as the number of weights increases, its advantages decrease. It needs lots of memory for the matrix operations. Special methods may decrease storage

requirements by the cost of growing time requirements. The modified LM described on p. 14 should have similar, even better, results.

Conjugate gradient methods perform well over a variety of problems. They are almost as fast as LM at function approximation and even faster on bigger networks. They are the best at pattern recognition and their performance does not degrade much when the error is reduced (remember the asymptotic stability request, see p. 12).

2.8 Generalization improvement

We have never a totally correct data, at least because computers work only with discrete numbers. Often there are some smaller or bigger errors in the data. When we have a too big network and learn it too long, it learns the examples so accurately that it learns all the errors. This problem is termed overfitting.

To tackle the problem of overfitting we can cut a part of the training data to form an evaluation set. These data can not be used as training examples, thus we can use this approach only when there are enough data. In the process of learning we calculate the error on the evaluation set after each epoch. When it increases, it indicates that the network has started to learn the examples' errors. At this point we stop the training — this is called early stopping.

In practice, we wait a given number of epochs, whether the error decreases or not, and remember the best weights configuration on which the error for the evaluation set was the least.

If the data is organized into series (see p. 20), we must assign whole series to the training or evaluation data, we can not split them.

Another possibility is to add noise to each training input pattern — a new random vector is added before the input feeds the network. For the network it becomes more and more difficult to fit the data precisely. This method is simple and computationally undemanding [12].

A similar result may be achieved by the iterative training approach (as opposed to the batch training, see p. 10) with random train patterns order.

2.9 Recurrent networks

Working with the data organized into series (see p. 20) involves using a recurrent network architecture and appropriate learning algorithms. The recurrent connections are fed from neuron outputs of the previous data row. For the

first data row in a series they are fed with some filler symbols (usually 0).

One of the training algorithms looks at the recurrent network as a big unfolded network formed by a series of networks where the recurrent connections form inputs to appropriate neurons in the next network. This approach involves a small change in the backpropagation algorithm. We run the network successively for all the rows in a series and store the neuron outputs of all the hidden and output units for each row. We feed the recurrent connections with outputs stored from the previous row. The error backpropagation starts from the last row in the series and goes back through the unfolded network. In each step we store the error function partial derivatives coming from the recurrent connections and add them in the next step (which runs on the previous row). Thus the neuron output becomes

$$a_i^q = f_i\left(\sum_{j \in i_{\leftarrow}^{\mathrm{fw}}} w_{ji} a_j^q + \sum_{j \in i_{\leftarrow}^{\mathrm{rec}}} w_{ji} a_j^{q-1}\right), \tag{2.28}$$

where $i_{\leftarrow}^{\mathrm{fw}}$ is the set of neurons forward-connected to j, $i_{\leftarrow}^{\mathrm{rec}}$ is the set of neurons with recurrent connections to j. The outputs a_i^{-1} are set to a filler value, usually 0.

The equation 2.11 for the output neurons remains the same. The equation 2.12 for the hidden neurons

$$\delta_j^q = \frac{\partial E^q}{\partial a_j} = \frac{\partial a_j^q}{\partial n_j} \sum_{r \in j^{\rightarrow}} \delta_r^q w_{rj} \text{ for } j \notin Y \tag{2.29}$$

splits into two equations — one calculates the recurrent connections gradient and stores it to be used in the next step

$$\delta_j^{q\ \mathrm{store}} = \frac{\partial a_j^{q-1}}{\partial n_j} \sum_{r \in j_{\mathrm{rec}}^{\rightarrow}} \delta_r^q w_{rj} \text{ for } j \notin Y, \tag{2.30}$$

the other uses the stored gradient and adds the forward connections gradient to it

$$\delta_j^q = \delta_j^{q+1\ \mathrm{store}} + \frac{\partial a_j^q}{\partial n_j} \sum_{r \in j_{\mathrm{fw}}^{\rightarrow}} \delta_r^q w_{rj} \text{ for } j \notin Y. \tag{2.31}$$

The recurrent error backpropagation is summarized in algorithm 5 (see p. 9 for symbols details). Here n_s means the length of series s.

Algorithm 5: Recurrent backpropagation

initialize the weights (see p. 8)
set the neuron outputs $a_i^{-1} = 0$
for all series $\boldsymbol{x}_s$
    for all training inputs $\boldsymbol{x}_s^q$ in the series, $q = 0, \ldots, n_s - 1$
        apply $\boldsymbol{x}^q$ to the input layer
        propagate $\boldsymbol{x}^q$ forward to the output layer, using eq. 2.28
    for all training inputs $\boldsymbol{x}_s^q$ in the series, $q = n_s - 1, \ldots, 0$
        calculate error $E^q(\boldsymbol{w})$ according to eq. 2.3
        compute the δ's of the output layer as in eq. 2.11
        compute the δ's of the preceding layers, by backpropagating
            the δ's backward as in eq. 2.30 and 2.31

Chapter 3

Training data

We can look on the training data as on a table with rows containing the training examples — inputs and their desired outputs. Each column in this table represents one input or output of all the training examples.

Sometimes we need to work with the data organized into series. My approach to form prosody by syllables looks on sentences as series of syllables. This allows two interesting possibilities: We can use a window technique and a recurrent network which uses previous results to compute the outputs (see p. 17).

When working with series we expect that the neighbour data have some relation to the current ones. The window technique takes the column value from a given number of left and right neighbours. For example one input tells whether a syllable is the first one in a word. Looking at the neighbours we know whether the previous or the next syllable was the first in some word. We have to fill the inputs with some filler symbols at the ends of the series (e.g. for the right neighbour of the last syllable).

Table 3.1 shows an example of two series of three rows with one column. After applying a window 1 to the left and 2 to the right and a filler symbol 0, the network inputs will be as in table 3.2.

In practice it is often advantageous to pre-process the data before feeding it to the network. Usually we want to transform them into a small range, e.g. $[-1; +1]$ or $[-3; +3]$. The simplest way is to find the minimum and maximum $x_{\min}$ and $x_{\max}$ of all the data in a column and apply

$$y = \frac{r_{\max} - r_{\min}}{x_{\max} - x_{\min}}(x + x_{\min}) + r_{\min}, \tag{3.1}$$

where $[r_{\min}; r_{\max}]$ is the desired range and y the transformed data x.

A more sophisticated way called input normalization finds the average μ

Table 3.1: Window example — source data

without the window	col 1
series 1, row 1	2
series 1, row 2	1
series 1, row 3	2
series 2, row 1	1.5
series 2, row 2	3
series 2, row 3	2

Table 3.2: Window example — result

with the window	l 1	col 1	r 1	r 2
series 1, row 1	0	2	1	2
series 1, row 2	2	1	2	0
series 1, row 3	1	2	0	0
series 2, row 1	0	1.5	3	2
series 2, row 2	1.5	3	2	0
series 2, row 3	3	2	0	0

and standard deviation σ of the data

$$\mu = \frac{1}{N}\sum_{n=1}^{N} x^n \tag{3.2}$$

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{n=1}^{N}(x^n - \mu)^2}, \tag{3.3}$$

where N is the number of data rows, and transforms them to

$$y = \frac{(x-\mu)\sigma_r}{\sigma} + \mu_r \tag{3.4}$$

where μ_r and σ_r are the desired average and standard deviation resp., usually 0 and 1 resp.

The perceptron construction allows it to transform any input in the same way: the input is multiplied by the weight and added to a bias. But using

Table 3.3: Multicategories

value	red	green	blue
green	0	1	0
red, green	1	1	0
yellow	0	0	0
blue	0	0	1

preprocessed data, the weights may remain small and the network training properties are better.

Not always are the data numerical, sometimes we work with categories like colors or sex. There are several ways to cope with them. We may assign a fraction of the destination range to each of them (0,$\frac{1}{n}$, $\frac{2}{n}$, ...), which will not be very useful. Or we can assign an order and code the order with a binary number, using $\log_2 n$ binary (0/1) inputs. We can fill one input with each category, than only 1 of the n inputs will have the value 1 and the rest 0. The last approach is especially useful when working with multi-categories, where every value may belong to several categories. When we meet an unexpected category, it feeds all the inputs with zeros. The table 3.3 shows an example of three inputs created for a category field with possible values red, green and blue.

Chapter 4

Training process

The network training process is very time consuming. Usually one has to try different sets of learning parameters and to look for some good ones. Every trial involves running the network training many times to try different weights initializations. All the configurable items like network architecture or training parameters are called degrees of freedom. The more degrees of freedom, the harder it becomes to try all their meaningful combinations. This chapter discusses the main ones: the hidden layer sizes, the input count, the training parameters and the learning goal.

It is difficult to set appropriate layer sizes. The smaller the network is the faster it runs and the better are its generalization abilities. But too small networks do not have enough parameters (weights, biases) to learn the given problem. There are several constructive algorithms which add neurons to the network steadily until it is big enough or which prune unnecessary connections and neurons, like cascade correlations [13].

I have tried a different approach — the training process starts with a static small network first, than it takes a bigger one and so on. When the network fitness does not improve any more, the smallest network was found. This involves a fitness function which runs for several initial weights configurations and takes the one with the smallest error.

The input count has a big influence on the network size and thus training time. Although in some cases the inputs are fixed, we may change them in some way. We may feed different number of inputs with categorial data, see p. 22. We may use a window of different sizes, see p. 20. And we can stop using some column at all.

The basic steepest gradient descent method (see p. 11) has its learning rate, its every improvement comes with another parameters. The stable conjugate gradient method (p. 12) and modified Levenberg-Marquardt (p. 14) have several parameters too. The difference is between the importance of

each parameter settings. Usually one parameter is driven by another one with smaller sensibility. Indeed, there is no need to change e.g. the stable conjugate gradient parameters at all. Although you can do it if you wish to experiment.

The particular network training is stopped when any of the conditions of the learning goal is filled:

- Error reached the desired value.
- Error has not improved for a given number of rounds. Usually we compare the error with the best one achieved and request some ϵ improvement. The number of rounds is important because if it is too small, the training will stop earlier than necessary, but when it is too big, we will waste time.

  If we are using an evaluation data set, the error on the evaluation set is compared instead of the training set error.
- Time used for the training is too long.
- Too many epochs passed.

I did use only the time and bad error improvement parameters, the others were set to values which were never achieved. The evaluation data set does help to avoid using them.

Chapter 5

Neural networks modelling prosody in specific languages

5.1 German

The Swiss TTS system SVOX [19] successfully uses a neural network to model prosody of indicative sentences.

The authors tried many different structures and the most successful one is a recurrent 62-20-10-8, i.e. with 62 inputs, 20 neurons in the first hidden layer, 10 neurons in the second and 8 outputs. The recurrent connections lead from 10 of the 20 neurons in the first hidden layer back to all the neurons in the first hidden layer.

The outputs are 8 F_0 (fundamental frequency) values. Each syllable is segmented into two demi-syllables. The middle for the training examples is set at the earliest intensity maximum. The two demi-syllables are split into two parts of equal length. For each of the resulting four segments the F_0 contour is approximated by a linear regression line, and the F_0 values at the line ends are the outputs.

The inputs are formed by two streams on which the window technique (see p. 20) is applied with different window sizes for each stream.

The first stream is formed by 5 bits (1/0 inputs). The coding is as follows:

- The first bit indicates an accent.
- The second bit indicates a phrase boundary with a non-terminal intonation (usually all but the last one in an utterance).
- The three remaining bits describe the phrase boundaries or the accent type.

- End of utterance is denoted (0,0,1,1,1).
- Word boundary is denoted (0,1,1,1,1).

Several accent types were considered [19, p. 175]: "pitch accents (accents associated with a major pitch movement), non-pitch accents on the main stress position in words, and secondary or tertiary word accents. All other syllables were judged unaccented."

The training corpus was manually phonologically transcribed. The phrase boundaries (start, end and pauses) are added to the transcription.

The second stream is formed by 4 bits:

- ShortV = 0 if the syllable contains a diphthong or a long vowel, 1 otherwise
- HighV = 1 if the syllable nucleus is a vowel with a generally high F_0, 0 otherwise
- LUC = 0 if no consonant occurs to the left of the syllable nucleus or the consonant to the left of the nucleus is voiced and quasi-stationary (the list of all such consonants is included in [19]), 1 otherwise
- RUC — like LUC, but for the consonant to the right

The streams are joined together. For phrase boundary symbols in the first stream, the values in the second are set all to 0. The window is 3 left and 6 right symbols on the first stream and 1 left and 1 right on the second. The window filler symbols are zeros.

Thus the network has $(3 + 1 + 6) * 5 = 50$ inputs from the first stream and $(1 + 1 + 1) * 4 = 12$ from the second, which is 62 together.

The algorithm used to train this network was steepest gradient descent (see p. 11) with recurrent backpropagation (see p. 17). The author used a learning rate of 0.02.

An example of the input streams: The SVOX-specific phonological representation of "Morgen kommt ein Gewitter" ("tomorrow comes a storm") is:

#{0}(P)[1]mqr-g6n#{2}(T)[3]kqmt <a_1n gq-[1]v1tqr#{0},

where (P) resp. (T) is a phrase without resp. with a terminal type intonation, #{0} and #{2} are the phrase boundaries, [1] and [3] are the accents.

The first stream of symbols will be formed by

#{0} 1 0 #{2} 3 $ 0 $ 0 1 0 #{0},

Table 5.1: Network input streams

First stream						Second stream			
symbol	inputs					ShortV	HighV	LUC	RUC
#{0}	0	1	0	0	1	0	0	0	0
1 Mor	1	0	0	0	1	1	0	0	0
0 gen	1	0	0	0	0	1	0	1	0
#{2}	0	0	0	1	1	0	0	0	0
3 kommt	1	0	0	1	1	1	0	1	0
word boundary	0	1	1	1	1	0	0	0	0
0 ein	1	0	0	1	1	0	0	1	0
word boundary	0	1	1	1	1	0	0	0	0
0 Ge	1	0	0	0	0	1	0	1	0
1 wit	1	0	0	0	1	1	1	0	0
0 ter	1	0	0	0	0	1	0	1	0
end of utterance	0	0	1	1	1	0	0	0	0

where the numbers stand for syllable accents and $ divides words. The resulting two streams are in table 5.1.

The author [19, p. 186] says about the network quality: "The difficulty of choosing a good network for F_0 prediction lies in the fact that the simple mean square distance measure between the natural and predicted F_0 contours is not a good indicator of the perceived naturalness of the generated contours. The networks had to be judged mainly by listening to the synthesized F_0 contours."

The author prepared some experiments in which the subjects had to indicate whether they listen to a natural or a TTS formed sentence. It seems that about 30% of the sentences sounded as natural as to be confused with natural contours.

The SVOX group has other speech corpus data ready allowing to train a network for questions and exclamations and a French speech corpus [15]. But the results with indicative sentences are still the best.

5.2 Korean

The authors in [11] propose a four-level model for synthesizing fundamental frequency F_0. The resulting F_0 is counted as a sum of all the levels. The paper does not contain enough information, thus I only try to reproduce the

basic ideas.

The first level is the global tune, a linear declination line $y = at + b$ where t is normalized by the number of words in a sentence. The parameters a and b are estimated by applying the least-square method on the training corpus.

The second level is the word pitch bias. It uses a statistical mapping table based on some 60 grammatical attributes. As there can not be all the combinations of the 60 attributes in the table, the closest combination is used.

The third level is the lexical tone and it uses a 2-layer neural network. The inputs are not well described in the paper.

The fourth level is the syllabic pitch pattern. Korean syllables consist of CVC, where initial and final C can be deleted. C means a consonant, V a vowel. In Korean, there exist 18 initial C, 21 V and 7 final C. A table assigning a pitch pattern to each syllable is used. The pitch is approximated by two straight lines which intersect at the syllable pitch maximum.

In this approach, the neural network had reportedly very good results. Its task was as simplified as possible by other statistical approaches.

Part II

Practical part

Chapter 6

Development environment

The program created within this thesis has a text-to-speech (TTS) part implemented in the Epos system and a neural networks part implemented in the Bang 3 system. The TTS part prepares the inputs for the network and works with its outputs. The neural networks part provides the network training process. Some of the Bang files are compiled with Epos to run a trained network.

Both Bang and Epos have some common features. Extensibility and portability of both is essential. The source code of both is written in some subset of C++ with minimal platform and operating system dependencies. Neither of them uses the Standard C++ Library (STL) nor the C++ streams. Most common C++ compilers on UNIX, Linux and MS Windows systems may be used for them. Both use the standard GNU autoconf tool to automatically check for system dependencies. Both provide a Microsoft Visual C++ 6 workspace under MS Windows.

6.1 Epos

The Epos TTS implementation [6, 5] is a highly configurable and flexible system. All the important options are user configurable without the need of recompilation. It is language independent, which means there are no source-level language-specific algorithms, though some features needed for other languages may be missing.

Epos is based on very general rules working with the internal text structure representation (see further). The rules are expressed in an explicit way, which is very useful for research purposes.

Although explicit rules are highly valued, some parts of the TTS process are not researched enough yet — especially prosody achieves better

Table 6.1: Unit levels

Level name	written TSR semantics	spoken TSR semantics
text	the whole text	the whole text
sent	sentence construction	terminated utterance
colon	sentence/clause/colon	intonational unit
word	word	stress unit
syll	word	syllable
phone	letter	sound
diphone		diphone

results with implicit trainable models like neural networks or generalized linear model [19, p. 148]. Epos supports combinations of rule-based and corpus-based methods like dictionaries with tags or neural networks [7, p. 31]. In its recent version it has some very limited support for neural networks. Within this thesis a much more flexible and powerful one was developed. Networks training is left outside Epos in the Bang system, but the interface is the same, even with the same source files, to ensure future compatibility.

The text to be spoken is internally stored in a format useful for application of transformational rules. This format is well suitable for neural network inputs forming too. Hanika says [5]:

"Every phonetic unit (or an approximation of one) is represented by a single node in the structure. The nodes are organized into layers corresponding to linguistic levels of description, such that a unit of level n can list its immediate constituents, that is units of level $n-1$. Every layer also has a symbolic name, which is used to refer to it in the rules."

The number and symbolic names of individual levels can be specified in the configuration files. They are currently defined [5] as in table 6.1. The level names do not have exactly the same meaning as in the common language. For example a `word` is a stress unit which in Czech may be formed by two words, e.g. "do kina" is one `word` because the preposition "do" is joined with the word "kina".

Recent stable Epos version 2.4 accompanies each segment in the TSR with values for all the prosodic parameters — linear interpolation is used between adjacent segments. This format limits prosody modelling to just one pitch point per segment and fixes the location of it, which is rather restrictive. Other TTS projects [11, 19] try to use more pitch points for each syllable to use all the information from the training corpus.

The prosody values — fundamental frequency F_0 or pitch, intensity and

duration — are all set as percentage of default values. Setting default F_0 to 100 Hz allows to use the values like it were frequencies in Hertz. The three values influence each other, for example a longer unit seems to be more intensive. There are some difficulties with processing the sound signal when the values differ much from the defaults. In the future Epos versions duration will be perhaps set in milliseconds, which will simplify the training data outputs creation. Now only F_0 seems ready to be formed by a network.

The Epos system does not use any CVS system (see p. 59), the changes are maintained by Jiří Hanika. It has a rather thorough user guide and some development guide [5].

A client called say communicates with the server by a text-to-speech control protocol [5]. All the text not parsed as options is converted to speech. A graphical user interface WinSay is available under MS Windows.

6.2 Bang 3

The distributed multi-agent system for building hybrid artificial intelligence models Bang 3, created at the Institute of Computer Science, Czech Academy of Science, Prague, is a follower of the Bang 2 and Bang 1 systems. It is a young project started at autumn 2001. The environment allows to run independent agents who communicate with each other. The agents may run on several computers in a cluster. Bang 3 will allow running the agents all in one process, with no communication overhead, or running each one in its own process, for easy debugging. In the current version only the one process mode is implemented.

To allow to write the code in an intelligible manner and at the same time include such powerful features like choosing a binary or textual data stream at run-time or allowing a transparent agent distribution over computer networks, Bang 3 uses heavily the C preprocessor and even a Perl script for the agents source files.

The agents communicate in an XML-derived language. They also can agree on a binary mode communication, but this is only a speed optimization — it always has its XML counterpart. The implementation decides when to use the binary communication. Each agent has its triggers which are evoked by matching events.

The main requirement for the agents is they support co-operative multitasking. When running a long lasting computation, they must split it into short iterations, call each iteration from the `idle` trigger and return control back to the Bang system.

The user interface is very limited at the moment. The user communicates

with the Text Console agent. When a key is pressed, the Text Console shows a prompt `>>` and expects a name of an agent followed by a message to be sent. For example:

```
>> Miluska request read file CString=nn.txt
```

This example shows a message in a simplified form: the `<`, `>` and `/` tags symbols are skipped, the closing tags and the attribute name `"value"` also. Every message not beginning with `<` is regarded as the simplified form. Every tag is placed as a subtag of the preceding one and the values following an equation sign `=` are quoted and added as a `value` attribute. The whole message is:

```
Miluska <request><read><file><CString value="nn.txt"/>
          </file></read></request>
```

In the agents description in the chapter 7 all messages are written in this simplified form.

The Text Console agent understands also two special commands: `halt` terminates the Bang system and `list` lists all the agents available.

The Bang 3 system has very few documentation of any kind. The Frequently Asked Questions are answered in the `doc` directory in the source distribution. Some development tips are displayed on the web pages of Pavel Krušina. In some aspects the documentation of the previous Bang 2 and Bang 1 systems included with the sources may be useful. Bang uses the sourceforge Internet project with its CVS [9] to maintain the source code.

Chapter 7

Bang agents

In this chapter you will learn about the three agents concerned with neural networks — `PerceptronNN`, `TrainingData` and `TrainingProcess` from the user point of view. Each of them has an XML-like user-editable configuration file which determines its function and a set of messages which it understands.

7.1 Configuration grammar

The configuration files contain XML tags. I do not use a standard DTD to explain the configuration syntax for two reasons. First the DTD does not describe the value types (number / string / boolean) and second I use two special tags `default` and `include`, see further.

The only attribute used in the configuration is `value`. All the variables are set by it. The configuration description in this thesis is simplified by omitting the `<`, `>` and `/` tags symbols, the closing tags and the attribute name `"value"`. The tag hierarchy is shown by left indentation. For example

```
file
    streams
        stream="$filename"
```

means
`<file><streams><stream value="$filename"/></streams></file>`.

Variables beginning with `$` are used in the description when a value needs a special description. The common ones are:

$filename file name with a path relative to the configuration file

$filename_with_percent like `filename`, but contains one `%` character which is replaced with a number to create a file name which does not

Table 7.1: Resolving the `default` tag

XML with `default`	will be resolved into

`columns`
`  default`
`  use="input"`
`  window`
`  left="2"`
`  right="2"`
`  column`
`  column`
`  use="output"`
`  window`
`  left="1"`
	`columns`
	`  column`
	`  use="input"`
	`  window`
	`  left="2"`
	`  right="2"`
	`  column`
	`  use="output"`
	`  window`
	`  left="1"`
	`  right="2"`

yet exist. When there is no `%`, the number is added at the end of the file name

$positive a positive integer number or 0
$string any string of characters
$float a real number

The optional tags are marked with `?`. For string variables this means that omitting the tag is the same as setting it to an empty string. For numerical and boolean (`"0"|"1"`) variables it is the same as setting it to `0`.

The tags which may be included once or more times are marked with `+`. The ones which may be included zero or more times are marked with `*`. For example:

```
?file
    streams
        +stream="$filename"
```

means the whole `file` tag may not appear in the configuration and the `streams` tag contains one or more `stream` subtags.

Two special tags may be used in the configuration files: the tag `default` adds its subtags to all the tags on the same hierarchy level. It does not replace existing subtags but adds all the new ones. An example is given in table 7.1.

The other special tag is `include="$filename"`, which is replaced by the contents of the respective file.

7.2 PerceptronNN agent

This agent provides two modes: in the run-only mode it only creates the structures needed to read weights and to run the network on given inputs to calculate outputs. In the training mode it prepares other structures needed for a chosen learning algorithm and runs that algorithm until any of the learning goal conditions is reached (see p. 24).

Apart of other training algorithms the network provides the weights perturbation method (see p. 11) to verify the correctness of the backpropagation, which is excellent for the debugging purposes. The weights perturbation is run after the gradient is calculated by the backpropagation. The agent prints the difference between this way computed derivatives and the backpropagation derivatives. The difference should be a very small number $\epsilon \ll 1$ because although the weights perturbation is a numerical method with a limited accuracy, its error is distributed around zero and thus by summing the errors for all the connections we get a very small number.

The configuration includes connection restrictions. These restrict the forward connections between neurons. The minimal requirement is the network being acyclic, which is achieved by creating only connections from neuron i to neuron $j, i < j$. There are no restrictions for the recurrent connections, because they use the values from the previous data row. Arbitrary recurrent connections may be created by the `recurrentConnections` tags. The neuron biases are included in the weights vector as the weights of connections from a fictive neuron -1.

The network architecture description includes further the count of neurons on each layer and the neuron transfer functions. These and the training and weights initialization algorithms are listed here:

Neuron forward connection restrictions:

layered connections from neurons on layer i to neurons on layer $i + 1$
none connections from neuron i to neuron $j, i < j$

Neuron transfer functions:

logsig logical sigmoid $f(x) = \frac{1}{1+e^{-x}}$
linear linear function $f(x) = x$
tansig hyperbolic tangent sigmoid $f(x) = \frac{2}{1+e^{-2x}} - 1$

Training algorithms:

gradientDescent fixed step steepest gradient descent, see p. 11. Uses the parameter `learningRate`

Table 7.2: Perceptron network run-only configuration

```
perceptronStructure
    connectionRestriction="layered"|"none"
    trainProcedure="runOnly"
    weights
        stream="$stream_value"
            type="local"|"remote"
            encoding="text"
            delimiter="$delimiter"
    layerTransferFuncs="$transferFuncs"
    layerSizes="$layerSizes"
    ?allRecurrentConnections
        +recurrentConnections
            start
                layer="$positive"
                ?from="$neuron"
                ?to="$neuron"
            end
                layer="$positive"
                ?from="$neuron"
                ?to="$neuron"
```

stableConjugateGradient stable conjugate gradient, see p. 12. Uses the parameters `learningRate`, `epsilon`, `zeta`, `c`

modifiedLM modified Levenberg-Marquardt, see p. 14. Uses the parameters `h`, `mi`, `mi_i`, `mi_d`

LM Levenberg-Marquardt, see p. 14. Uses the parameters `mi`, `mi_i`, `mi_d`

runOnly no training — the network is in the run-only mode

Weights initialization algorithms:

random random number from $[-1/\sqrt{|j_{\leftarrow}|}; 1/\sqrt{|j_{\leftarrow}|}]$ with normal distribution, where $|j_{\leftarrow}|$ is the count of all the neurons connected to the neuron j

nguyen-widrow Nguyen-Widrow initialization as described on p. 8. Not yet implemented with recurrent networks

The grammar of the `PerceptronNN` configuration for both modes is in the tables 7.2 and 7.3. One configuration may contain tags from both. The variables used there are (see p. 34 for the common variables):

Table 7.3: Perceptron network learning mode configuration

```
perceptronStructure
    connectionRestriction="layered"|"none"
    trainProcedure="gradientDescent"|"stableConjugateGradient"
        |"modifiedLM"|"LM"
    ?runWeightsPerturbation value="0"|"1"
    ?weightInitProcedure value="random"|"nguyen-widrow"
    layerTransferFuncs="$transferFuncs"
    layerSizes="$layerSizes"
    ?allRecurrentConnections
        +recurrentConnections
            start
                layer="$positive"
                ?from="$neuron"
                ?to="$neuron"
            end
                layer="$positive"
                ?from="$neuron"
                ?to="$neuron"
    params
        learningRate="0.002"
        ?epsilon="1e-005"
        ?zeta="1.5"
        ?c="0.5"
        ?h="0.005"
        ?mi="0.001"
        ?mi_i="2"
        ?mi_d="0.6"
    ?dumpWeights="$filename_with_percent"
    ?dumpWeightsDelimiter="$delimiter"
    ?dumpAllInterval="$positive"
    ?dumpAll="$filename_with_percent"
    ?logErrors="$filename"
```

string **$stream_value** is explained further
string **$delimiter** string of characters each of which is used as a column delimiter. You may use `"tab"` in place of the tabulator character
int **$neuron** positive numbers mean a neuron index in the given `layer` (the 1st neuron index is 0), negative numbers a percent of all the neurons in the layer
string **$layerSizes** the layer sizes delimited by `-`, e.g. `"15-8-4-1"`. The first and last size are replaced by the input and output counts resp. Thus `"0-8-4-0"` means the same
string **$transferFuncs** the layer transfer functions delimited by `-`, their count should be 1 less than the layer sizes count, e.g. `"logsig-logsig-linear"`

Tags used in the `PerceptronNN` configuration:

runWeightsPerturbation should the weights perturbation be run as described above?
weights when `dumpWeights` is empty, the weights are placed `locally` in `$stream_value`, otherwise they are placed externally in a file the name of which is created from `dumpWeights` and written into `$stream_value`.
recurrentConnections recurrent connections going from `start` to `end` to be created
params for different training algorithms different parameter sets are used (see p. 36)

The `PerceptronNN` agent understands these messages important for a Bang 3 user:

request read file CString=$filename reads a configuration file
request join TrainingData CString=$name
joins the `TrainingData` agent identified by his name. This agent will give the training data
request run starts the learning
request pause pauses the learning
request continue continues the learning
request end terminates the learning
request info displays information about the state of the learning

7.3 TrainingData agent

This agent reads the data files, examines them, sets filters (see further) and divides the rows between the training and evaluation sets. He works with

Table 7.4: Data splitting

inputs1.txt	outputs1.txt
data2.txt	

the training data as with a table. The rows of the table contain the training examples — the inputs and their desired outputs. Each column represents one input or output of all the training examples.

The data may be split horizontally and / or vertically. You can use more data files which contain whole rows and are added one after another (vertically) or which have different column subsets of the same data rows (horizontally). This may be useful when you prepare the network outputs in another way than the network inputs.

Table 7.4 shows an example of data splitting: first 100 rows are in two files — inputs in `inputs1.txt` and outputs in `outputs1.txt`, and next 100 rows are in one file `data2.txt`.

The input and output columns are processed by filters. A filter takes some numerical or textual value and translates it into one or more floating point numbers. Every input column is passed through a filter to form the network inputs — this is termed data pre-processing. The outputs are post-processed through a filter to get the desired value. The post-processing is inverse to the pre-processing. The agent is able to set the filters automatically by examining the data and finding the category values, the minimum, maximum, average and standard deviation in each column.

The filter types and the possible translations are listed here:

Filter types:

bool boolean — two different values, e.g. 0 / 1, yes / no
int integer number
float floating point number
category category values, interpreted as text even if numerical
multicategory category values, each row contains any number of them (0–n) — not yet implemented

Filter translations:

float linearly transforms as in eq. 3.1

$$y = \frac{r_{\max} - r_{\min}}{x_{\max} - x_{\min}}(x + x_{\min}) + r_{\min},$$

linear normalizes by average and standard deviation as in eq. 3.4

$$y = \frac{(x - \mu)\sigma_r}{\sigma} + \mu_r,$$

where μ is named `avg` and σ is named `stdev` in the configuration file

bools for categories or multicategories only: assigns one boolean 0 / 1 input to each category contained in the training data. The number of these network inputs equals the number of categories. For categories not contained in the training data all these inputs will equal 0. The categories may be listed in the `categories` tag

none do not translate, use values as they are

The grammar of the `TrainingData` configuration is in the table 7.5. The variables used there are (see p. 34 for the common variables):

string **$delimiter** string of characters each of which is used as a column delimiter. You may use `"tab"` in place of the tabulator character. The row delimiter is the end-of-line

int **$range** positive numbers mean a row index (the 1st row index is 0), negative numbers a percent of the row count. When working with series, a series index or a percent of the series count is meant

string **$categories** semicolon (`;`) separated list of categories

Tags used in the `TrainingData` configuration:

trainingDataFiles files split horizontally (into several `file` tags) and vertically (into several `stream` tags)

examineColumns should the columns be examined to set the filters — `categories, minR, maxR, stdev, avg`?

dumpCfg used by the `TrainingProcess` agent to dump the configuration after examining the columns

ranges assigns parts of the data to the training and evaluation data sets

columns describes each column usage - as an `input`, `output` or not used at all

window may be used only when the data are organized into series. The column values of `left` preceding and `right` following rows in the same series will be added to the inputs. When there are no such rows in the series (e.g. no row is preceding the first one), the `empty` input is used. See p. 21 for an example

minR, maxR, avg, stdev the filter settings

series when greater than zero, the data is organized into series

seriesSeparator the series in the data files are separated by rows beginning with the `seriesSeparator`

Table 7.5: Training data configuration

```
trainingData
    ?trainingDataFiles
        +file
            +streams
                +stream="$filename"
                    type="remote"
                    delimiter="$delimiter"
                    encoding="text"
    ?examineColumns="0"|"1"
    ?dumpCfg="$filename_with_percent"
    ranges
        +range
            type="train"|"eval"
            ?from="$range"
            ?to="$range"
            ?rest="0"|"1"
            ?random="0"|"1"
    columns
        +column
            use="input"|"output"|"no"
            type="bool"|"int"|"float"|"category"
                |"multicategory"
            translate="float"|"linear"|"bools"|"none"
            categories="$categories"
            ?minR="$float"
            ?maxR="$float"
            ?avg="$float"
            ?stdev="$float"
            ?window
                ?left="$positive";
                ?right="$positive"
                ?empty="$float"
    ?series="$positive"
    ?seriesSeparator="$string"
```

The `TrainingData` agent understands these messages important for a Bang 3 user:

request read file CString=$filename reads a configuration file and all data

request info displays information about the number of the data in the training and evaluation sets

7.4 TrainingProcess agent

The `TrainingProcess` agent guides the whole process of training. It goes through a predefined set of batches. Each batch may work with other training data and perceptron network structure. For each batch the fitness of the network is calculated by trying several initial weights configurations and choosing the one with the smallest error. The number of `trials` is set in the configuration file.

The training process is able to find the smallest network architecture by trying growing network architectures. It multiplies the layer sizes in each step by a given number `findSmallest` and adds 1 if the layer size does not change (e.g. when multiplying $1 * 1.4$). At each step it computes the fitness as described above. When the fitness ceases to improve, the batch is finished.

A log file is created which allows to analyse the results. It contains a header with information about the training and evaluation data size, network architecture and names of files with the particular `TrainingData` and `PerceptronNN` configuration. Under the header there is one row for each network training trial with layer sizes, connection restrictions (see p. 36), best error achieved on the training and evaluation sets, number of epochs, time passed, training procedure used, learning goal condition (see p. 24) which terminated the training, and the neuron transfer functions.

The grammar of the configuration file is in the table 7.6. Tags used there are:

logName the log file described above

dumpNN the trained network ready to be used is saved into this file with all its weights and with the training data information

startBatch allows to continue in an interrupted training process. Find the last `batch` index value in the log file

learningGoal controls when will be the network training stopped, see p. 24: if the network error reaches `error` or if the time passed is longer than `time` or if the epochs count is bigger than `epochs` or if the error has

Table 7.6: Training process configuration

```
trainingProcess
    ?logName="$filename"
    ?dumpNN value="$filename_with_percent"
    ?startBatch value="$positive"
    learningGoal
        error="$float"
        time="$positive"
        epochs="$positive"
        minErrorImprovement="$float"
        badErrorImproveEpochs="$positive"
    batches
        +batch
            perceptronStructure ...
            trainingData ...
            ?findSmallest="$float"
            trials="$positive"
            ?stopOnSuccess="0"|"1"
```

not improved at least of `minErrorImprovement` for `badErrorImproveEpochs` epochs

batch contains the whole `perceptronStructure` and `trainingData` configurations (which may be placed in external files by the `include` tag, see p. 35)

findSmallest if non-zero, the training process tries to find the smallest network architecture as described above

trials how many times will each network be run with different weights to find its fitness. If set to 0, no network is ever run

stopOnSuccess should I stop when the desired `error` value (set in `learningGoal`) was reached?

The `TrainingProcess` agent understands these messages important for a Bang 3 user:

request read file CString=$filename reads a configuration file

request join TrainingData CString=$name
joins the `TrainingData` agent identified by his name

request join PerceptronNN CString=$name
joins the `PerceptronNN` agent identified by her name. This agent will

train the neural network and send a message to the `TrainingProcess` agent when it is ready

request run starts the training process

request end terminates the training process

request info displays information about the state of the process

Chapter 8

Epos neuralnet rule

The neural network feature is included into the Epos system by a new rule in the language-specific configuration file `prosody.rul` (see Epos documentation [5] about rules). The syntax is

```
neuralnet filename sent syll
```

The real configuration file name replaces *filename*. The word `sent` tells Epos that the scope of the rule is the unit on level `sent` (see p. 31 for unit levels). The word `syll` is the target level of the rule. Both may be replaced by other level names.

The configuration is an enriched `TrainingData` configuration. The Epos-specific tags are listed in the table 8.1. The variables used there are (see p. 34 for the common variables):

string **$epos_input** an input expression described further

Epos-specific tags used in the `TrainingData` configuration:

char2floats definitions of user defined functions as described further

epos_output the output may be placed as the fundamental frequency F_0 or just ignored

epos_loglevel when `use="no"`, all the contents of the phone units contained in the unit on this level are printed

seriesSeparator if set, the series in the `epos_log` will be separated by it. The series are formed by the scope level of the `neuralnet` rule (which is `sent` in our case)

epos_nn the `PerceptronNN` configuration file used by Epos. When empty, Epos is only preparing the training data: it does not try to run the network

Table 8.1: Epos-specific training data configuration

```
trainingData
    ?char2floats
        +char2float="$string"
            *chars="$float"
                src="$string"
            ?default="$float"
            ?empty="$float"
    columns
        +column
            use="input"|"output"|"no"
            ?epos="$epos_input"
            ?epos_output="frequency"|"none"
            ?epos_loglevel="text"|"sent"|"colon"|"word"
                |"syll"|"phone"|"diphone"
    ?seriesSeparator="$string"
    ?epos_nn="$filename"
    ?epos_log="$filename"
    ?epos_traindata="$filename"
```

Table 8.2: User `char2float` function example

```
char2float="sonority"
    chars="1" src="$diphtong"
    chars="0.9" src="$long"
    chars="0.8" src="$short"
    chars="0.7" src="$sonant$SONANT"
    chars="0.6" src="$voiced"
    default="0.5"
    empty="0"
```

epos_log the network inputs and outputs are written into this file which can later be used as a training data file

epos_traindata the `TrainingData` configuration file used by the network

8.1 Inputs

Each input is described as an expression with Epos Text Structure Representation (TSR) specific functions and user defined `char2float` functions.

The `char2float` functions assign values to characters. They may be applied on any unit, because in the Epos TSR each unit has a one character content. Phones hold their phoneme, sentences their end mark etc.

The functions are defined by a `char2float` tag, which may contain one `default` subtag, one `empty` subtag and any number of `chars` subtags. An example is given in the table 8.2.

You may use predefined Epos variables in the `src` tag (see [5] for details on variables defining). These are denoted by a name preceded by `$` and contain a set of characters, for instance `$voiceless` contains `ptťkfsšcčxř`. No character may be assigned in more than one `chars` tags. The `default` value is given to all the units not listed in any of the `chars` tags. The `empty` value is given to empty units, which may be reached by a `next` or `prev` function — for example `prev` from some first unit.

The priority of the arithmetical and logical functions is the same as in the C language, see table 8.3. Operators with higher priority are evaluated earlier. All the operators on the same priority level are processed from left to right.

This means that for example `2 + 2 * 5` will be processed as `2 + (2 * 5)` as `+` has a lower priority than `*`, and `2/3/4` as `(2/3)/4`. In many cases

Table 8.3: Operator priority

lowest priority
`OR`
`AND`
`== !=`
`< <= > >=`
`+ -`
`* /`
`!`
highest priority

the default priority will satisfy your needs. You can use the parentheses `()` if it does not or if you are not sure.

The inputs grammar is described in a little simplified form in table 8.4. I did not enclose the parenthesis nor the commas into the quotes. The symbol `char2float` stands for a name of a user defined char to float function, `char2float_quoted` is a name enclosed in quotes. I did not describe the possibility to omit any of the three `next` and `prev` functions parameters.

In all the functions working with unit levels (see p. 31) when you use a lower level than the target level (set in the `neuralnet` rule), the result is undefined. For instance if the target level were `word`, you could not use `count("phone","syll")`. The functions available are:

basic arithmetical functions `+, -, *, /`

basic logical functions comparison `<, <=, >, >=`, not-equals `!=`, equals `==`, the negation `!`, the logical `AND` and `OR`. For example:

```
count("phone","syll") > 2 AND count("phone","syll") <= 5
```

All the non-zero numbers are converted to the logical true, zero is converted to the logical false. When converting a logical value to a number, true becomes 1 and false 0

char2float functions `float` *`char2float`* `(unit)` where *`char2float`* is replaced by the name of a user-defined function, returns the value as described above

index `int index (level1, level2)` returns the index of the unit on the `level2` inside the unit on the `level1`. For example:

Table 8.4: Grammar of neural network inputs

```
input = float

unit = "next" (unit, level, int)
     | "prev" (unit, level, int)
     | "this"
     | "ancestor" (unit)
     | "maxfloat" (char2float_quoted, level, level)
     | (unit)

float = char2float (unit)
      | "!" float
      | float binary_operator float
      | int
      | (float)
      | FLOAT_NUM

int = "count" (level, level)
    | "index" (level, level)
    | "basal_f" (unit)
    | "cont" (unit)
    | INT_NUM

binary_operator = "OR" | "AND" | "==" | "!=" | "<" | "<="
                | ">" | ">=" | "+" | "-" | "*" | "/"

INT_NUM = [1-9][0-9]*

FLOAT_NUM = {-}INT_NUM{"."INT_NUM}

level = "text"|"sent"|"colon"|"word"|"syll"|"phone"|"diphone"
```

`index ("word", "sent")` is the index of the word containing the current processed unit inside the sentence

count `int count (level1, level2)` returns the count of the units on the `level2` inside the unit on the `level1`. For example:

`count ("phone", "syll")` is the count of the phones inside the syllable containing the current processed unit

next `unit next (unit, level, count)` returns the unit on the `level` which is `count` to the right of the current processed unit. You can omit any of the parameters, defaults are `unit = this, level = target level, count = 1`. For example:

`next ("word",2)` is the syllable 2 to the right from the current word, when the target level in the `neuralnet` rule is `syll`

prev like `next`, but move to the left

this `unit this` returns the current processed unit on the target level

ancestor `unit ancestor (level)` returns an ancestor unit of the current one. It is the same as `next (level, 0)`

maxfloat `unit maxfloat(char2float,level1,level2)` returns the unit on the `level2` with maximum value assigned by the user defined `char2float` function from all the units inside the scope `level1`. For example:

`maxfloat("sonority","phone","syll")` finds the most sonorous phone in the syllable if `sonority` is defined as in table 8.2

basal_f `int basal_f (unit)` returns the fundamental frequency F_0 calculated by preceding rules outside this `neuralnet` rule

cont `int cont (unit)` returns the ASCII code of the one character content of the given unit, see p. 48

Chapter 9

Neural network controlling prosody

This chapter describes the process of creating the particular neural network included with this Thesis.

The network forms prosody of indicative Czech sentences in a neutral tone. It was trained by a speech corpus consisting of 18 sentences of various length (see Appendix C). The sentences were read by 3 speakers, one of which (Pavel Machač) read them twice. The desired outputs in the training data were extracted from Machač's 36 utterances. The training data contain 858 rows, one for each syllable.

The fundamental frequency F_0 contour is found by pitch pulses. The sampling frequency was 8 kHz, which means for a time distance $T = (pp_i - pp_{i-1})/8000$ between two pitch pulses $F_0 = 1/T = 8000/(pp_i - pp_{i-1})$. For unvoiced segments of the utterances the F_0 values are assigned by a linear interpolation between the neighbour voiced sections. An example of the F_0 contour is given in the table 9.1.

The sound borders in the time dimension were extracted manually, by listening to parts of the sentence sound file. This work lasts many hours. An example of the results is given in the table 9.2. Both the F_0 and the phone borders were extracted by a team of Petr Horák at the Institute of Radioelectronics using a Horák's own created software Speech Studio (`http://sstudio.ure.cas.cz`).

The training data inputs were created by the log file feature of the Epos `neuralnet` rule (see p. 46). I have prepared a C++ script which joined together the F_0 and the phone borders to compute an average F_0 for the phones. It than found the syllables in the Epos log and assigned the F_0 of the most sonorous phone in the syllable to them.

The inputs selection was based on consultations with Jiří Hanika, to get

Table 9.1: Pitch pulses

	frame	$F_0 = 8000/pp_i - pp_{i-1}$
pp_1	4049	
pp_2	4141	87
pp_3	4233	87
pp_4	4326	86
pp_5	4421	84.2
pp_6	4517	83.3
pp_7	4616	80.8
pp_8	4718	78.4
pp_9	4823	76.2
pp_{10}	4928	76.2
pp_{11}	5021	86
pp_{12}	5102	98.8
pp_{13}	5180	102.6

some phonological foundation. An advantage of using a neural network is that we only have to know *what* phonological properties control the prosody and not *how* nor *how much*. The inputs are:

count("syll","word") == 1 monosyllabic `words`
count("syll","word") count of `syllables` in the `word`
index("syll","word") / count("syll","word") order in $[0;1]$ of the `syllable` in the `word`
index("word","colon") == 1 AND count("word","sent") > 1 the initial `word`, but not an isolated one
index("word","colon") == count("word","colon") AND count("word","sent") > 1 the terminal `word`, but not an isolated one
count("word","colon") count of `words` in a `colon`
index("word","colon") / count("word","colon") order in $[0;1]$ of the `word` in the `colon`
index("syll","word") == 1 the first syllable is stressed in the Czech language

The `TrainingProcess` agent in Bang creates configuration files, each of which contains the network description, the weights and the training data description. Setting the file name in the Epos `neuralnet` configuration, one can immediately start to use the new trained network.

Table 9.2: Phone borders for the sentence "Vlak dnes nejede."

start frame	end frame	phone
3982	4982	V
4982	5388	L
5388	5851	A
5851	6584	G
6584	7369	D
7369	7701	N
7701	8135	E
8135	9161	S
9161	9918	N
9918	10324	E
10324	11063	J
11063	11465	E
11465	12276	D
12276	13297	E

I have experimented with various network and inputs sizes. I have used a window of 0 to 4 neighbours both left and right. This means with 8 Epos inputs the network had 8, 24, 40, 56 or 72 inputs. From the 36 utterances 24 formed the training data and 12 the evaluation data. The learning algorithm was the stable conjugate gradient, see p. 12. All the networks had recurrent connections leading from half of the neurons in the first hidden layer to all the neurons in the first hidden layer.

Table 9.3 shows the resulting error on the training and evaluation data sets. The frequency values in the training examples had the average 99.1981 and the standard deviation 20.3818 and the column was filtered with the `linear` translation, see p. 40. Thus the numbers in Hertz in the table are post-processed from the sum-squared-error E_{SSE} to $20.3818\sqrt{E_{\mathrm{SSE}}}$. The post-processed error is a geometrical average of the errors for the training patterns.

The columns in the table contain connection count, layer sizes, minimal post-processed error in Hertz on the evaluation set and on the training set, average error ± standard deviation on the evaluation and training sets and count of trials with the same architecture and different initial weights.

The errors do not differ much depending on the network size — the smallest network with only 10 connections achieved the best error 11.0 while the largest one with 1167 connections 8.5. That is perhaps because of the evalua-

tion set technique: the training was stopped when the error on the evaluation set began to grow.

There are some common problems with prosody generated by neural networks. As the prosody qualities are random in some extent, the network may never learn perfectly. And the simple numerical error result may not be the main criterion — we must listen to the prosody formed by the network to judge its quality [19, p. 186]. To tackle this problem, Traber proposes [19, p. 196]: "One of the most useful prerequisites for future research on F_0 prediction by means of neural networks would be a perceptually appropriate, computable measure of the deviation of synthetic contours from natural contours. Such a measure could be used for the training as well as for the evaluation of F_0 predictors."

Hailichová [4] was extremely precise in assessing quality, but she had an inconsistent speech corpus and she mixed together two kinds of questions, which have a very different melody in the Czech language — the so-called "yes-no" and "wh-" questions.

As the post-processed error is a rather big number around 10 Hertz, the networks differ much in the way they achieve it. However, listening to the prosody formed by the networks some common properties can be found. The prosody is much more lively than the one currently used in Epos formed by rules. Most networks learned well the speaker's behavior at the sentence ends, where he strongly lowers his voice.

Table 9.3: Training results ordered by connection count

con. count	layer sizes	eval error	train error	eval avg	train avg	trials
10	8-1-1	11.0	10.5	11.1 ± 0.05	10.5 ± 0.06	20
20	8-2-1	10.4	9.9	10.9 ± 0.20	10.2 ± 0.18	20
21	8-2-1-1	10.4	9.7	10.9 ± 0.29	10.3 ± 0.30	25
26	24-1-1	9.9	9.1	10.0 ± 0.07	9.3 ± 0.06	40
32	8-3-1	10.1	9.1	10.6 ± 0.25	9.8 ± 0.32	20
36	8-3-2-1	9.8	9.2	10.5 ± 0.32	9.7 ± 0.43	22
42	40-1-1	9.9	8.8	9.9 ± 0.04	8.9 ± 0.04	20
44	8-4-1	9.6	8.7	10.3 ± 0.27	9.4 ± 0.40	20
52	24-2-1	9.2	8.2	9.6 ± 0.14	9.0 ± 0.21	40
53	24-2-1-1	9.4	8.6	10.0 ± 0.23	9.2 ± 0.24	28
55	8-4-3-1	9.8	8.8	10.2 ± 0.25	9.2 ± 0.35	20
58	56-1-1	9.6	8.2	9.8 ± 0.05	8.4 ± 0.05	20
72	8-6-1	9.4	8.4	10.1 ± 0.32	8.9 ± 0.34	20
74	72-1-1	9.8	8.1	9.9 ± 0.04	8.2 ± 0.03	14
80	24-3-1	8.7	7.6	9.4 ± 0.25	8.5 ± 0.49	40
84	40-2-1	9.0	7.6	9.5 ± 0.26	8.2 ± 0.33	20
84	24-3-2-1	9.4	8.3	9.8 ± 0.19	8.8 ± 0.35	13
85	40-2-1-1	9.1	7.6	9.5 ± 0.33	8.0 ± 0.46	10
94	8-6-4-1	9.6	7.7	10.0 ± 0.40	8.5 ± 0.67	10
108	24-4-1	8.9	7.1	9.4 ± 0.19	8.2 ± 0.55	40
116	56-2-1	9.0	6.8	9.4 ± 0.28	7.5 ± 0.34	20
117	56-2-1-1	9.1	7.3	9.5 ± 0.31	8.0 ± 0.51	14
119	24-4-3-1	9.3	7.9	9.6 ± 0.23	8.7 ± 0.48	10
122	8-9-1	9.0	7.2	9.7 ± 0.43	8.1 ± 0.64	20
128	40-3-1	8.8	7.0	9.3 ± 0.25	7.7 ± 0.51	20
132	40-3-2-1	9.0	6.7	9.2 ± 0.28	7.3 ± 0.63	8
168	24-6-1	9.0	7.2	9.3 ± 0.16	7.8 ± 0.43	20
172	40-4-1	8.6	6.5	9.1 ± 0.21	7.2 ± 0.48	20
172	8-9-6-1	9.2	6.1	9.7 ± 0.42	7.8 ± 1.06	10
176	56-3-1	8.5	6.0	9.1 ± 0.30	6.8 ± 0.56	20
180	56-3-2-1	8.5	6.3	9.2 ± 0.46	7.4 ± 0.92	10
190	24-6-4-1	8.9	6.8	9.4 ± 0.22	8.0 ± 0.52	12
202	8-13-1	8.8	5.8	9.4 ± 0.46	7.4 ± 1.12	20
236	56-4-1	8.7	5.4	9.1 ± 0.26	6.4 ± 0.77	20
247	56-4-3-1	8.5	5.9	9.0 ± 0.33	6.7 ± 0.90	10
264	40-6-1	8.7	5.6	9.0 ± 0.21	6.6 ± 0.67	20
314	8-13-9-1	8.9	5.8	9.4 ± 0.33	6.9 ± 0.80	10
316	24-9-6-1	8.7	6.1	9.2 ± 0.29	7.6 ± 0.73	10
352	8-19-1	9.0	5.1	9.6 ± 0.47	7.1 ± 1.06	20
522	24-13-9-1	8.8	6.6	9.2 ± 0.21	7.6 ± 0.71	5
592	8-19-13-1	8.7	5.2	9.0 ± 0.35	6.3 ± 1.01	10
1167	8-28-19-1	8.5	5.1	8.9 ± 0.28	5.4 ± 0.28	9

Chapter 10

Summary

This thesis describes the perceptron network learning process. Three agents in the multi-agent system for artificial intelligence Bang 3 participate on it: the `PerceptronNN` agent contains the learning algorithms, the `TrainingData` agent pre- and post-processes the data and the `TrainingProcess` agent tries various network architectures and input sizes. The agents work with time series, too. The recurrent network architecture and the window technique using the inputs from the neighbour training data examples are implemented.

The outputs of the training process are network configuration files and a log file. Each configuration file contains the network description, its weights and the training data description. The log file describes the resulting error size and allows to analyse the network learning and to choose the most suitable network.

The future research may add some other learning algorithms like the genetics ones and another network architectures like the RBF networks. Also some generalization of the fitness functions would be useful, giving the possibility of experimenting with any learning parameters in a similar way to the process of finding the smallest network described in this work.

The other software part of this thesis is a very flexible configuration file in the text-to-speech system Epos allowing to describe the inputs to a neural network and the usage of its outputs. Several neural networks controlling prosody generation are created using a small training corpus consisting of 18 Czech indicative sentences read twice in a neutral voice. The networks compute the fundamental frequency F_0 for each syllable.

The prosody formed by the neural network is much more lively than the current one in Epos formed by rules. Various networks produce very different sentence melodies sounding more or less natural. Better results can be expected using a bigger training corpus.

An interesting direction of future research would be some perceptually

appropriate, computable measure of the deviation of the synthesized contours from the natural ones. The simple sum-squared-error computed for each syllable does not estimate the practical network quality.

Appendix A

Terminology

Bang a multi-agent system for artificial intelligence

CVS Concurrent Version System — a dominant open-source network-transparent version control system

DTD Document Type Definition — defines the legal building blocks of an XML document. Defines the document structure with a list of legal elements

epoch one round of network training — the network is run on all the training examples and the weights are changed by a learning algorithm

Epos a highly configurable Text-To-Speech system for the Czech language, one of the best nowadays

fundamental frequency F_0 the basic frequency with which a sound producing object vibrates. In this work it is used as the fundamental frequency of the vocal cords

norm the Euclidean norm

$$|\boldsymbol{x}| = \sqrt{\sum_i x_i^2}$$

perceptron the basic unit of a perceptron neural network. Its output is a weighted sum of its inputs on which a simple quick transfer function is applied, see p. 3

performance function the same as an error function, usually the SSE, see p. 8

positive semidefinite a matrix $\boldsymbol{A}$ is said to be positive semidefinite if $\boldsymbol{v}^{\boldsymbol{T}}\boldsymbol{A}\boldsymbol{v} > 0$ is true for any non-zero vector $\boldsymbol{v}$

prosody a set of speech properties — usually pitch, intensity and duration

RBF Radial Basis Function networks — neural networks with units forming circles around the unit center rather than dividing space into subspaces like perceptrons

singular a matrix is singular if the vectors forming its rows are not linearly independent or (equivalently) if the inverse matrix does not exist

SSE Sum-Squared Error — sum of squares of differences between the desired and the real outputs, see p. 8

STL Standard Template Library — a C++ library of container classes, algorithms, and iterators included as a part of the C++ norm but not yet implemented on all systems in full scope

TSR Text Structure Representation — the internal structure of the Epos system, see p. 30

TTS Text-To-Speech — the process of synthesizing speech from written texts by computers

Appendix B

Author's own source files contribution

The Bang 3 sources of the classes `CPerceptronNN` and `CTrainingData` are stored apart of the agents definitions to allow to use the classes outside the Bang system.

In the Bang 3 source distribution, the directory `mod/neuralnet` contains the three agents `PerceptronNN`, `TrainingData` and `TrainingProcess`:

nnagents.h declares the `PerceptronNN` and `TrainingData` agents

perceptron.b `PerceptronNN` agent Miluska and her triggers — the agent forms an interface to the `CPerceptronNN` class methods

perceptron.cc run-only mode methods of the `CPerceptronNN` class — creating the neurons and connections, running the network, copying the outputs

perceptron.h declares the `CPerceptronNN` class as inherited from the `TPerceptronStructure` class

percinit.cc weights initialization methods of `CPerceptronNN`

percstruct.cc input / output methods for the `TPerceptronStructure` and `CPerceptronNN` classes

percstruct.h declares the `TPerceptronStructure` class which contains the network configuration and is used by the `TrainingProcess` to manage the network configuration

perctrain.cc perceptron learning related methods — weights perturbation, backpropagation, modified Levenberg-Marquardt, stable conjugate gradient, the methods which iterate through the learning process

traindata.b `TrainingData` agent Igor and his triggers — the agent forms and interface to the `CTrainingData` class methods

traindata.cc `CTrainingData` methods — data reading, examination, input window processing, iterating through the data table

traindata.h declares the `CTrainingData` class and defines its inline functions. Declares the `TFilter` class with its inline pre- and post-processing functions

trainprocess.b `TrainingProcess` agent John and his triggers. As opposed to the other two agents, he is not only an interface but contains all the training process functionality — iterates through the training batches, calls the other two agents to read the data and train a network and creates the training log

trainprocess.h declares the `TrainingProcess` agent and his structures

xmltempl.cc the three agents configuration reading / writing

xmltempl.h templates for the three agents configuration

In the Bang 3 source distribution, in the directory `ext`:

utils.h,cc a few useful utilities

xmlutils.h,cc XML reading / writing utilities, used by the `xmltempl.cc`

In the Bang 3 source distribution, in the directory `dict` there are the basic data structures. I have created some new containers based on the previous existing ones and inspired with the C++ Standard Template Library (STL):

matrix.h,cc number matrix. Supports matrix multiplication and inversion

pair.h,cc pair of two arbitrary types

set.h,cc contains a sorted data vector

string.h,cc this file was created by Pavel Krušina, I have added some features — `replace` to replace substrings by another ones, `split` to split into parts by a given delimiter, `substr` to get a substring

svector.h,cc small vector is an alternative vector class to the `TVector` created by Pavel Krušina with smaller memory requirements

iterator.h `iterator` and `const_iterator` allow to unify the iteration through a vector, a set and a map, the second one gives access to constant objects

xmlstream.h,cc a structure allowing to read / write containers in an alternative, short manner without the XML tags around each item. The network weights are written and the training data are read by it

The Epos source files are all in the directory `src`:

neural.y Bison parser which creates an expression tree from a neural network input definition

neural.h,cc structures used to execute the `neuralnet` rule: `CNeuralNet` is the main class which prepares the inputs, runs the network and places the outputs, `CExpression` contains and computes the expression tree, `TTypedValue` encapsulates a union to allow to work safely with various types

Appendix C

Network Training Corpus

The training corpus prepared by Petr Horák was formed by 2 x 2 x 3 x 3 sentences: There are compound sentences consisting of 1, 2 or 3 sentences and these may be short or long. Every sentence type is represented by 3 different sentences. Every sentence was spoken twice by Pavel Machač. Both utterances were used.

1. Vlak z Pardubic do Plzně dnes odpoledne nejede.
2. Jiné podrobné údaje o kulturních akcích nemáme.
3. Soubor obsahující tabulky dat je uložen.
4. Vlak dnes nejede.
5. Jiné údaje nemáme.
6. Soubor je uložen.
7. Trať z Pardubic do Plzně se tento týden opravuje, a proto osobní vlak ve čtrnáct padesát nejede.
8. Výsledek našeho výpočtu nebude dost přesný, protože zatím nemáme k dispozici jiné přesné údaje.
9. Soubor obsahující tabulky dat je uložen a program bude po stisknutí klávesy Enter ukončen.
10. Trať se opravuje, a proto vlak dnes nejede.
11. Výsledek nebude přesný, protože nemáme jiné údaje.
12. Soubor je uložen a program bude ukončen.

13. Trať z Pardubic do Plzně se tento týden opravuje, a proto je dočasně omezen provoz mezi oběma místy, dokud nebudou odstraněny následky prudkého deště.

14. Výsledek posledního výpočtu nebude tak přesný, jak jsme při zahajování výzkumného úkolu chtěli, protože v současné době nemáme jiné přesné údaje.

15. Soubor obsahující úplné tabulky dat je uložen, program bude ukončen stiskem klávesy Enter a počítač se po uložení ostatních programů vypne.

16. Trať se opravuje, a proto je provoz omezen a vlak dnes nejede.

17. Výsledek nebude tak přesný, jak jsme chtěli, protože nemáme jiné údaje.

18. Soubor je uložen, program bude ukončen a počítač se vypne.

Appendix D

User guide

This appendix describes the programs included on the CD with this thesis. It is the Epos system and the Bang 3 system, both fully functional on MS Windows and many Linux and Unix distributions. Please read the file `readme.txt` for last updates and changes.

Both systems are included in one archive `eposbang.zip` because the source files in Epos have a relative path to the source files in Bang. To run Bang 3 on a Unix / Linux, run these files:

```
cd bang/bang3
autogen.sh
configure
ln -s impl.DivineOffering impl
make
bang3
```

To run Bang 3 on a Windows system, you must install ActivePerl into the directory `Program Files\Perl`. You can download it from web, but for you convenience it is included on the CD. Run the file `bang\bang3\win32ize.bat` and open the bang3.dsw workspace in MS Visual C++ 6, build the files and run Bang from the environment.

When you press any key, a prompt `>>` is shown. See p. 32 for details. Two special commands are defined to run the training process agent to train a network. The command `train` runs the training process
`bang/bang3/test/manual/ffnn/trproc.txt`
which uses the corpus for the prosody network. It is the sequence of the following messages:

```
John request read file CString=test/manual/ffnn/trproc.txt
John request join TrainingData CString=Igor
John request join PerceptronNN CString=Miluska
```

```
John request run
```

The command `train1` runs the training process `trproc_test1.txt` with a training corpus consisting of 40 pairs of a random number $x \in [-1; +1]$ and $y = 0.5 + 0.25 \sin(3\pi x)$. You can change the learning algorithm used and other parameters in the configuration files. The training log and the trained networks are stored in `bang/bang3/test/manual/ffnn/results`.

During the training you can ask the training process `John`, the neural network `Miluska` or the training data `Igor` for information, e.g.

```
Miluska request info
```

You can stop the training by `John request end` and quit Bang 3 by `halt`.

To run Epos on a Unix/Linux, first configure Bang. Than run the files:

```
cd epos/epos/src
make install
epos --base_dir ../cfg
say some-text-to-be-spoken
```

To run Epos on Windows, run `epos/epos/arch/win/configure.bat`. Open the `src/epos.dsw` workspace in MS Visual C++ 6 and set the active configuration to `say - Win32 Release` in menu `Build`. Than build the project. Run the file `epos.bat` in the main directory. The built client is in `release/bin/say.exe`.

The prosody configuration files are in `epos/epos/cfg/lng/czech`. The file `prosody.palkova` contains the prosody rules used in the public Epos versions. If you copy it to `prosody.rul`, you can listen to the prosody formed without neural networks. The file `prosody.neu` contains the configuration file used for the prosody network. By changing the `epos_nn` value in it you can listen to various networks.

Bibliography

[1] Jiří Šíma a Roman Neruda. *Teoretické otázky neuronových sítí*. Matfyzpress, Praha, 1996.

[2] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press Inc., 1997.

[3] Howard Demuth and Mark Beale. *Neural Network Toolbox (For Use with Matlab), User's Guide Version 4*. The Mathworks, Inc., 2000.

[4] Hana Hailichová. Řízení základního kmitočtu syntetické řeči pomocí neuronové sítě. Master's thesis, ČVUT FEL, Praha, 1998.

[5] Jiří Hanika. Epos on-line documentation. `http://epos.ure.cas.cz`.

[6] Jiří Hanika. Text-to-speech synthesis. Master's thesis, MFF-UK, Praha, 2000.

[7] Jiří Hanika and Petr Horák. Dependences and independences of text-to-speech. In Hans-Walter Wodarz, editor, *Forum Phoneticum 69*, Frankfurt a.M., 2000.

[8] Pavel Jiroušek. Sestavování prstokladů pro houslový part. Master's thesis, MFF-UK, Praha, 1998.

[9] Pavel Krušina. Bang on-line documentation. `http://bang2.sf.net`.

[10] Petra Kudová. Neuronové sítě typu rbf pro analýzu dat. Master's thesis, MFF-UK, Praha, 2001.

[11] Jung-Chul Lee, Youngjik Lee, Sang-Hun Kim, and Minsoo Hahn. Intonation processing for tts using stylization and neural network learning method. `http://www.asel.udel.edu/icslp/cdrom/vol3/433/a433.pdf`.

[12] Jiřina Marcel. *Neuronové sítě - skripta*. UK MFF KSI, Praha, 1995.

[13] Ondřej Maštálka. Neuronové sítě a jejich využití pro zpracování a predikce časových řad. Master's thesis, MFF-UK, 2001.

[14] I. Petrović, M. Baotić, and N. Perić. An efficient newton-type learning algorithm for mlp neural networks. `http://www.rasip.fer.hr/act/papers/NC98_913-085.PDF`.

[15] B. Pfister, V. Jantzen, and C. Traber. Verbesserung der natürlichkeit in der sprachsynthese, jahresbericht 2000 für das projekt cost258. Zürich, 2001.

[16] R. Salomon. *Verbesserung konnektionistischer Lernverfahren, die nach der Gradientenmethode arbeiten*. PhD thesis, TU Berlin, October 1991.

[17] R. Salomon and J. L. van Hemmen. Accelerating backpropagation through dynamic self-adaptation. *Neural Networks*, 1996.

[18] D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors. *Advances in Neural Information Processing Systems*, pages 225–231. Number 8. MIT Press, Cambridge, 1996.

[19] Christof Traber. *SVOX: The Implementation of a Text-to-Speech System for German*. PhD thesis, Swiss Federal Institute of Technology, Zürich, 1995.